Causality analysis
A type-based representation

Marc Pouzet

ENS

Marc.Pouzet@ens.fr

Workshop SYNCHRON
Bamberg
22 Nov. 2024

Mutually recursive definition of streams

E.g., take Lustre, Scade, Lucid Synchrone, FRP, Zelus, Simulink, etc.

An old question:

> *Is the system reactive/productive/deadlock-free, that is, at every instant and for any valid input, does it produce an output?*

Semantically: every reaction is the result of a fix-point computation starting with $\bot$ (deadlock).

A main system is causally correct (dynamically) when, provided inputs are $\neq \bot$, outputs are $\neq \bot$.

Causality is a semantical property: it depends on the semantics of primitives.

E.g., interpreting `if`/`then`/`else` strictly or lazilly, the `or` to be strict, sequential or parallel leads to different causalities.

Causality analyses

The causality analysis computes a static over-approximation.

To ensure that the final main system is (dynamically) causal.

Causality analysis in synchronous programs

The simplest is that of Lustre [5, 9]: build a dependence graph; check the graph is acyclic.

Esterel and Signal compilers implemented a more expressive analysis which involves boolean reasonning.

E.g., Two equations in parallel [1]

```
(* [c, i1, i2] are inputs *)
x = if c then i1 else y and y = if c then x else i2
```

Lustre semantics: It is not causal because `if/then/else` is strict, that is, given an input environment $[c_0/c, v_1/i_1, v_2/i_2]$, lfp is $[\bot/x, \bot/y]$.

Signal semantics: It is causal because `if/then/else` is lazy, that is, given $[c_0/c, v_1/i_1, v_2/i_2$, lfp is $[v/x, v/y]$ with $v = mux(c_0, v_1, v_2)$.

[1] written in Zelus syntax.

Written in Esterel

The program is rejected by a Lustre compiler; it is accepted by a Signal compiler.

Its Esterel version is semantically causally correct too.

It is accepted and compiled by an Esterel compiler that follows [3].

```
[present c then
  [present i1 then emit x]
   else
     [present y then emit x]
] ||
[present c then
  [present x then emit y]
   else
     [present i2 then emit y]
]
```

Causality analysis in synchronous programs

None of the causality analyses in Lustre, Esterel, Signal was modular.

A modular causality analysis was done for LS [8].

I can be expressed as a type system; it is implemented in Scade 6 [7] and Zelus [4].

The analysis is unconditional: an expression either (always) depend on an input or never.

Two examples in Esterel

The program P13 from the Esterel primer V5.91.
`https://github.com/marcpouzet/zrun/blob/work/tests/good/esterel-primer-p13.zls`

```
(* Section 5.1.4 *)
module P13:
  input I;
  output O1, O2;
  present I then
    present O2 then emit O1 end
  else
    present O1 then emit O2 end
  end present
end module
```

A naive, graph-based analysis, would reject it because of an apparent cycle (`o1` depends on `o2`; `o2` depends on `o1`).

Nonetheless, it is causal and is accepted by Esterel compilers.

The program P14 from the Esterel primer V5.91.
https://github.com/marcpouzet/zrun/blob/work/tests/good/esterel-primer-p14.zls

```
module P14:
  output O1, O2;
  present O1 then emit O2 end;
  pause;
  present O2 then emit O1 end
 end module
```

It is causal and is accepted by Esterel compilers.

Expressed in Lustre, P13 is not causal.

```
node p13(i) returns (o1, o2)
  let
   (o1, o2) =
    if i then if o2 then (true, false) else (false, false)
    else if o1 then (false, true) else (false, false);
  tel;

node p13(i) returns (o1, o2)
  let
    o1 = if i then if o2 then true else false else false;
    o2 = if i then false else if o1 then true else false;
  tel;

node p13(i) returns (o1, o2)
  let
    o1 = i & o2;
    o2 = not i & o1;
  tel;
```

Starting with $[\bot/o_1, \bot/o_2]$ and, for any value v for i, the least solution for the equation that defines o_1 and o_2 is $[\bot/o_1, \bot/o_2]$.

But...

P13 can be written in Scade/Zelus using a by-case definition of streams:

```
node p13(i) returns (o1 default false, o2 default false)
  if i then
   do if o2 then o1 = true done
  else
   do if o1 then o2 = true done
```

which is a short-cut notation for:

```
node p13(i) returns (o1, o2)
  if i then
   do if o2 then o1 = true else o1 = false
   and o2 = false done
  else
   do if o1 then o2 = true else o2 = false
   and o1 = false done
```

The two are causal for trivial reasons, without any boolean reasonning or complicated machinery.

By-case definitions [6]

Suppose that E_1 and E_2 are two equations. E_1 define the set of variables N_1; E_2 defines the set of variables N_2.

The by-case definition `if/then/else` defines variables in $n \in N_1 \cup N_2$.

If an equation is missing for n in a branch, n gets a default value. Either that given at the declaration; or its previous value (`n = last n`).

```
local x default 42, y do
if c then x = f(y) else y = g(x)
```

means:

```
local x, y do
if c then do x = f(y) and y = last y done
else do y = g(x) and x = 42 done
```

The by-case construction generalizes to other control structures, e.g., automata. The causality analyses exploits the by-case definitions of streams.

Two classical examples that are causally correct but are rejected by the compilers of Lustre, Scade, Zelus.

The cyclic circuit of Malik

https://github.com/marcpouzet/zrun/blob/work/tests/good/malik.zls

```
let node mux(c, x, y) returns (o) if c then o = x else o = y

let node f(x) returns (o) o = 2 * x
let node g(x) returns (o) o = x - 1

let node fog_gof(c, x) returns (y)
  local x1, x2, y1, y2
  do x1 = mux(c, x, y2)
  and x2 = mux(c, y1, x)
  and y1 = f(x1)
  and y2 = g(x2)
  and y = mux(c, y2, y1)
  done

(* It is equivalent to *)
let node fog_gof_reference(c, x) returns (y)
  y = mux(c, g(f(x)), f(g(x)))
```

The cyclic token ring arbiter

`https://github.com/marcpouzet/zrun/blob/work/tests/good/arbiter.zls`

Constructive causality

```
(* Constructiveness in the sense of Esterel *)
(* Verbatim from The Esterel Primer, V5.91, Berry, 2000
*-
*- 1. An unknown signal can be set present if it is emitted.
*- 2. An unknown signal can be set absent if no emitter
      can emit it.
*- 3. The then branch of a test can be executed if the
      test is executed and the signal is present.
*- 4. The else branch of a test can be executed if the
      test is executed and the signal is absent.
*- 5. The then branch of a test cannot be executed if the
      signal is absent.
*- 6. The else branch of a test cannot be executed if the
      signal is present.
*)

(* Moreover, Esterel makes a special treatment of the two
   boolean operators
*- (or and &) that are considered parallel and not sequential.
*)
```

The problem we consider

Some equations deadlock like `x = x + 1` or `x = y and y = x`.

A language with only length-preserving stream functions.

- Lifting: lift a scalar into a constant stream; a n-ary function to apply pointwise.
- A unit delay, initialized or not. E.g., `(x : xs) fby ys = x : ys`.
- function definition and application, possibly higher-order.

Detect and reject stream equations that are not productive, i.e., ensure that all streams are infinite

Several works address the question of productivity and proof techniques for languages manipulating infinite data structures.

Hamming's exercise in SASL. [Dijkstra, 1981]

On the productivity of recursive list definitions. [Sijtsma, 1989].

Proving the correctness of reactive systems using sized types. [Hugues & Pareto & Sabry, 1996];

Guarded recursion in proof assistants. E.g.,:

Infinite objects in type theory. [Coquand, 1993];

Structural recursive definitions in type theory. [Gimenez, 1994];

Termination checking in the presence of nested inductive and coinductive types. [Danielson, Altenkirch, 2010];
Beating the Productivity Checker Using Embedded Languages. [Danielson, 2010];
(Many others: Abel, Bertot, Buchholz, Di Gianantonio & Miculan, Hancock & Pattinson & Ghani, McBride, Morris & Altenkirch & Ghani, etc.)

Related works

Those works consider a more general language where stream functions can be length preserving or not and/or mixed with inductive structures.

E.g: is the following equation productive?

```
x = 0 : 1 : tl x
```

where `tl (x : xs) = xs`

We only have an operator "delay" that make streams longer but not shorter. [2]

We adress a simpler problem: we forbid to write `tl` which is not length preserving.

[2] `tl` can be defined by `tl x = x when (false fby true)`. `when` is not a length preserving function.

Operators

Hence, the language has essentially the following features:

1. Define mutually recursive equations;
2. point-wise application of an operations (e.g., $+$);
3. unit delay: `pre`, `fby`.
4. The non length preserving operators: `when` and `merge` are considered as if they were length preserving, from the causality analysis point-of-view.

A trivial solution

Build a dependence graph from the syntax such that:

- For every equation $x = e$, state that x depends on all variables appearing in e but those on the right of a unit delay (`pre` or `fby`).
- compute the transitive closure;
- reject recursive definitions if the corresponding graph is cyclic.

This solution is easy to implement. It accepts:

```
let node int(x') = x where
  rec x = 0 fby (x' + 1)

let node fix(g)(x0) = x where
  rec x = g(x0 fby x) in x
```

But rejects:

```
let node f(x) = (y, z) where
  rec (y, z) = let t = x + 1 in (z, t)

let node copy(x, y) = (x, y)

let node main(x) = (t, u) where
  rec = copy(x, t)
```

It is very sensitive to naming and the syntactic structure. It does not treat modularity — the ability to define a function, compute some information about it once and reuse it later.

We propose a *type based representation of input/output dependences*.

The idea of representing causality information as a type was firt introduced in the language Lucid Synchrone [8].

This is the way it is done in Scade [7] but for a first-order language only.

Here, we go a bit further by considering higher order with a new formulation of dependences and algorithm for type simplification.

A preliminary solution was published in [2]. What is presented today is implemented in the Zelus compiler since 2017 but was never published.

Examples in Zélus [3]

[3] zelus.di.ens.fr

Summary

- Represent the instantaneous input/output dependence by a type
- A stream expression is associated to a tag.
- A partial order between those tags.

```
let node f(x, y, z) = x + y, y + z

let node forward_euler(t)(k, x0, u) = output where
  rec output = x0 -> pre (output +. (k *. t) *. u)

let node backward_euler(t)(k, x0, u) = output where
  rec output = x0 -> (pre output) +. (k *. t) *. u

let node filter(n)(h)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = forward_euler(h)(n, 0.0, udot)

val f : {'a < 'b , 'c}. 'c  * 'a  * 'b -> 'c  * 'b
val forward_euler : {}. 'a -> 'a  * 'b  * 'a -> 'b
val backward_euler : {}. 'a -> 'a  * 'a  * 'a -> 'a
val filter : {}. 'a -> 'b -> 'b  * 'b -> 'b
```

```
  let node bad_filter(n)(h)(k, u) = udot where
    rec udot = n *. (k *. u -. f)
    and f = backward_euler(h)(n, 0.0, udot)
```

```
File "examples.zls", line 17, characters 10-41:
>  and f = backward_euler(h)(n, 0.0, udot)
>          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Causality error: This expression has causality type
'c, whereas it should be less than 'd
Here is an example of a cycle:
f at 'd < udot at 'c; udot at 'c < f at 'd
```

Higher order

A function can have an argument which is a function.

```
let node gfilter(int)(h)(n)(k, u) = udot where
  rec udot = n *. (k *. u -. f)
  and f = run (int(h)) (n, 0.0, udot)

let node gpid(int)(filter)(h)(p, i, d, u) = c where
  rec c_p = p *. u
  and i_p = run (int h)(i, 0.0, u)
  and c_d = run (filter h)(d, u)
  and c = c_p +. i_p +. c_d

val gfilter :
{'a < 'b}. ('c -> 'a  * 'd  * 'b -> 'b) -> 'c ->
           'a -> 'b  * 'b -> 'b
val gpid :
  {'a < 'b}.
    ('c -> 'd  * 'e  * 'a -> 'b) ->
    ('c -> 'f  * 'a -> 'b) -> 'c -> 'b  * 'd  * 'f  * 'a -> 'b
```

Higher order

```
let node filter_forward(h)(n)(k, u) =
  generic_filter(forward_euler)(h)(n)(k, u)

val filter_forward : {'a < 'b}. 'b -> 'a -> 'a * 'a -> 'a

(* This program is not causal *)
(* let node filter_backward(h)(n)(k, u) =
  generic_filter(backward_euler)(h)(n)(k, u) *)
```

```
>  gfilter(backward_euler)(h)(n)(k, u)
>          ^^^^^^^^^^^^^^^^
Causality error: This expression has causality type
'c -> 'd * 'e * 'f -> 'g, whereas it should be less than
'h -> 'i * 'j * 'k -> 'l
Here is an example of a cycle:
k < f; f < g; g < l; l < k
```

Remove administrative relations. Then, `f < g` and `l < k` are contradictory

A language kernel

Definition of functions; variables, constant, application, fix-point, tuples and access functions.

$$
\begin{array}{rcl}
d & ::= & \texttt{let}\ f\ x = e \mid d; d \\
\\
e & ::= & x \mid v \mid \texttt{let rec}\ x = e\ \text{in}\ e \\
& & \mid (e, e) \mid \texttt{fst}(e) \mid \texttt{snd}(e) \\
& & \mid e(e) \mid e\ \texttt{fby}\ e
\end{array}
$$

v stand for values.

Typing constraints so that $\texttt{let rec}\ x = e\ \text{in}\ e'$ is limited such that the type of e is bounded: it has no function type in it.

Expressing Dependences/Causality with a type

Since all stream operations are length preserving, express instantaneous dependences only.

The dependence relation is a partial order.

Represent the instantaneous dependences of an expression by a type.

$$
\begin{array}{lcl}
bt & ::= & \alpha \\
t & ::= & bt \mid t \times t \mid t \to t \\
\sigma & ::= & \forall \alpha_1, ..., \alpha_n : C.t \mid t \\
\\
C & ::= & \{\alpha_i < \alpha_j\}_{i,j \in I}
\end{array}
$$

$\alpha_1, ..., \alpha_n, ...$ are tags (a tag is a "time stamp").

C must define a partial order (acyclic graph) between those tags.

Initial conditions

$$
\begin{array}{lcl}
(+) & : & \forall\alpha.\alpha \times \alpha \to \alpha \\
\texttt{if . then . else .} & : & \forall\alpha.\alpha \times \alpha \times \alpha \to \alpha \\
\texttt{pre}\ \cdot & : & \forall\alpha_1, \alpha_2 : \{\alpha_2 < \alpha_1\}.\alpha_1 \to \alpha_2 \\
\cdot\ \texttt{fby}\ \cdot & : & \forall\alpha_1, \alpha_2 : \{\alpha_1 < \alpha_2\}.\alpha_1 \times \alpha_2 \to \alpha_1
\end{array}
$$

The typing predicate: $C, H \vdash e : t$, where:

$H = [x_1 : \sigma_1, ..., x_n : \sigma_n]$ and $Acyclic(C)$ as an implicit side condition.

(fundef)

$$\frac{C, H[x : t_1] \vdash e : t_2}{H \vdash \texttt{let}\ f\ x = e : H[f : Gen(C)(t_1 \to t_2)]}$$

(app)

$$\frac{C, H \vdash f : t_1 \to t_2 \qquad C, H \vdash e : t_1}{C, H \vdash f\ e : t_2}$$

(const)

$$C, H \vdash v : bt$$

(tuple)

$$\frac{C, H \vdash e_1 : t_1 \quad C, H \vdash e_2 : t_2}{C, H \vdash (e_1, e_2) : t_1 \times t_2}$$

(var)

$$\frac{C_x, t \in \mathit{Inst}(\sigma)}{C + C_x, H[x : \sigma] \vdash x : t}$$

(sub)

$$\frac{C, H \vdash e : t_1 \quad C \models t_1 < t_2}{C, H \vdash e : t_2}$$

(rec)

$$\frac{C, H[x : t] \vdash e : t_e \quad C \models t_e < t \qquad C, H[x : t_e] \vdash e' : t'}{C, H \vdash \texttt{let rec}\ x = e\ \text{in}\ e' : t'}$$

Generalisation

$$\mathit{Gen}(C)(t) \quad = \quad \forall \alpha_1, ..., \alpha_n : C.t \ \text{where} \ \mathit{Vars}(t) = \{\alpha_1, ..., \alpha_n\}$$
$$\text{provided } \mathit{Acyclic}(C)$$

Instanciation

$$C[\vec{\alpha'}/\vec{\alpha}], t[\vec{\alpha'}/\vec{\alpha}] \in \mathit{Inst}(\forall \vec{\alpha} : C.t) \text{ provided } \mathit{Acyclic}(C[\vec{\alpha'}/\vec{\alpha}])$$

The dependence/precedence order

The relation is strict.

(tuple)
$$\frac{C \models t_1 < t_1' \qquad C \models t_2 < t_2'}{C \models t_1 \times t_2 < t_1' \times t'2}$$

(fun)
$$\frac{C \models t_2 < t_2' \qquad C \models t_1' < t_1}{C \models t_1 \to t_2 < t_1' \to t_2'}$$

(trivial)
$$C[\alpha_1 < \alpha_2] \models \alpha_1 < \alpha_2$$

(trans)
$$\frac{C \models t_1 < t_2 \qquad C \models t_2 < t_3}{C \models t_1 < t_3}$$

We write $C \models H < H'$ iff $\forall x. C \models H(x) < H'(x)$

Equations

Extend the language of equations.

$$\begin{array}{rcl} e & ::= & \texttt{let rec}\ E\ \texttt{in}\ e \mid \ldots \\ E & ::= & E\ \texttt{and}\ E \mid \texttt{if}\ e\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \\ & & \mid x = e \mid \texttt{local}\ x\ \texttt{in}\ E \end{array}$$

(eq)
$$\frac{C, H \vdash e : t}{C, H \vdash x = e : [t/x]}$$

(and)
$$\frac{C, H \vdash E_1 : H_1 \qquad C, H \vdash E_2 : H_2}{C, H \vdash E_1\ \texttt{and}\ E_2 : H_1 + H_2}$$

(rec)
$$\frac{C, H \vdash E : H' \qquad C \models H' < H \qquad C, H + H' \vdash e : t}{C, H \vdash \texttt{let rec}\ E\ \texttt{in}\ e : t}$$

(local)
$$\frac{C, H + [t/x] \vdash E : H' + [t'/x] \qquad C \models t' < t}{C, H \vdash \texttt{local}\ x\ \texttt{in}\ E : H'}$$

By-case definition

$H_1 + H_2$ is the union of H_1 and H_2. Domains must not intersect.
$H_1 \# H_2$ merges two environments. $(H_1.H_2)(x) = t$ if $H_1(x) = H_2(x) = t$ or $H_1(x) = t$ and $x \notin Dom(H_2)$ or $H_2(x) = t$ and $x \notin Dom(H_1)$.

(if)
$$\frac{C, H \vdash e : \alpha \qquad C, H \vdash E_1 : H_1 \qquad C, H \vdash E_2 : H_2}{C, H \vdash \texttt{if } e \texttt{ then } E_1 \texttt{ else } E_2 : H_1.H_2}$$

but it rejects $\texttt{if } c \texttt{ then } y = f(x) \texttt{ else } x = g(y)$

(if)
$$\frac{C, H \vdash e : \alpha \qquad C, H, H_1, H_2'' \vdash E_1 : H_1' \qquad C, H, H_2, H_1'' \vdash E_2 : H_2'}{C, H \vdash \texttt{if } e \texttt{ then } E_1 \texttt{ else } E_2 : H_1'.H_2'}$$

where $H_1' < H_1 < H_1''$, $H_2' < H_2 < H_2''$

Simplification of constraints

Sub-typing constraints have to be simplified.

The type system for causality is similar to a type system with intersection and union types.

- $t_1 < t \wedge t_2 < t$ corresponds to $t_1 \cup t_2 < t$;
- $t < t_1 \wedge t < t_2$ corresponds to $t < t_1 \cap t_2$.

The current system do not have relations of the form $\alpha < t$ or $t < \alpha$, where t is not a variable.

Causality is done after typing. It uses the type structure to construct causality skeleton types.

Yet, a type scheme $\forall \alpha_1, \alpha_2, \alpha_3 : \{\alpha_1, \alpha_2 < \alpha_3\}.\alpha_1 \times \alpha_2 \to \alpha_3$ is equivalent to $\forall \alpha.\alpha \times \alpha \to \alpha$.

Type simplification for systems with intersection/union types has been studied a lot, in particular by Aiken & Wimmers, Pottier, Smith & Trifonov, Castagna et al.

Input/Output relation

We apply the simplification algorithm that uses the Input/output relation of Pouzet & Raymond [10].

$InOut(p)(t)$ computes the set of inputs and outputs. $p \in \{-, +\}$ is a polarity. $neg(-) = +$ and $neg(+) = -$.

$$
\begin{array}{lcl}
InOut(+)(\alpha) & = & \emptyset, \{\alpha\} \\
InOut(-)(\alpha) & = & \{\alpha\}, \emptyset \\
InOut(p)(t_1 \to t_2) & = & \mathit{let}\ i_1, o_1 = InOut(neg(p))(t_1)\ \mathit{in} \\
 & & \mathit{let}\ i_2, o_2 = InOut(p)(t_2)\ \mathit{in} \\
 & & i_1 \cup i_2, o_1 \cup o_2 \\
InOut(p)(t_1 \times ... \times t_n) & = & \mathit{let}\ (i_k, o_k = InOut(p)(t_k))_{k \in [1..n]}\ \mathit{in} \\
 & & \cup_{k \in [1..n]} i_k, \cup_{k \in [1..n]} o_k
\end{array}
$$

Given a set of variables V and a set of constraints C between them.
$I \subseteq V$ the set of inputs; $O \subseteq V$ the set of outputs. I and O not necessarily disjoint.

- $Out(a) = \{b \in O \mid C \vdash a \leq b\}$
- $In(a) = \{b \in I \mid C \vdash b \leq a\}$
- $IO(a) = \{b \in I \mid Out(a) \subseteq Out(b)\}$

For every input and output variable, computes its IO set.

Associate a unique key (a fresh variable) to every IO set.

Replace the relation $<$ by the relation between IO sets, that is:

if $IO(a) \subseteq IO(b)$, *with* a' *the key of* $IO(a)$ *and* b' *the key for* $IO(b)$, *then* $a' < b'$.

There is a canonical form (i.e., unicity) that minimises the number of variables.

Extra simplification

Some dependences can be removed.

Only keep dependences of the form $\alpha^p < \beta^q$ where the polarities p is $-$ or $+-$ and q is $+$ or $+-$.

It gives extremely short type signature in practice.

Open question: does it simplify more than existing simplification methods for type systems with sub-typing constraints?

Conclusion

- The type system is used since 2017 in the Zelus compiler.
- Try it! `https://github.com/INRIA/zelus`
- The compiler of Scade 6 also implements a type-based causality analysis.
- Boths take into account by-case definitions of streams.

Extra notes (to be continued).

The curse of non linear functions

```
let node fix(f)(x) = o where rec o = run f (x, o)
let node twice(f)(x) = o where rec o = run f (run f (x))
let node twice(f)(x) = o2 where
  rec o1 = run f (x) and o2 = run f (o1)
val twice : {'a < 'b}. ('b -> 'a) -> 'b -> 'a
```

The type of `twice` says that the output of `f` must not depend on its input whereas it does not appear in any recursive stream equation!
As a consequence, we cannot write:

```
let node f(x) = x + 1
let node main(x) = twice(f)(x)

>let node main(x) = twice(f)(x)
>                           ^^^
Causality error: This expression has causality type
'b -> 'c, whereas it should be less than 'd -> 'e
Here is an example of a cycle:
d < b; b < c; c < e; e < d
```

Atomic functions

This is a consequence of the contravariance rule and the fact that the sub-typing rule uses a strict order.

Would it be better using union/intersection types? a strict and non strict order?

One way to impose the strongest constraints on an input function is to consider it to be atomic, that is, as if all of its outputs would depend on all of its inputs.

```
let node twice_atomic(f)(x) = o where
  rec o = run (atomic f) (run (atomic f) (x))

let node twice_atomic_f(x) = twice_atomic(f)(x)

val twice_atomic : {'a < 'b}. ('a -> 'b) -> 'b -> 'b
val twice_atomic_f : {}. 'a -> 'a
```

The typing rule for atomic values

$$
\begin{array}{lcl}
\texttt{skeleton}(C)(\alpha)(\alpha') & = & \alpha, C + [\alpha < \alpha'] \\
\\
\texttt{skeleton}(C)(\alpha)(t_1 \to t_2) & = & \mathit{let}\ t_1', C_1 = \texttt{skeleton}(C)(\alpha')(t_1)\ \mathit{in} \\
 & & \mathit{let}\ t_2', C_2 = \texttt{skeleton}(C_1)(\alpha)(t_2)\ \mathit{in} \\
 & & t_1' \to t_2', C_2 + [\alpha' < \alpha] \\
\\
\texttt{skeleton}(C)(\alpha)(t_1 \times t_2) & = & \mathit{let}\ t_1, C_1 = \texttt{skeleton}(C)(\alpha)(t_1)\ \mathit{in} \\
 & & \mathit{let}\ t_2, C_2 = \texttt{skeleton}(C_2)(\alpha)(t_2)\ \mathit{in} \\
 & & t_1' \times t_2', C_2
\end{array}
$$

$$
\frac{t', C' = \texttt{skeleton}(\emptyset)(\alpha)(t)}{C + C', H[f : t] \vdash \texttt{atomic}\ f : t'}\ (\text{atomic})
$$

References I

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet.

A Type-based Analysis of Causality Loops in Hybrid Systems Modelers.

In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.

Albert Benveniste, Benoit Caillaud, Hilding Elmqvist, Khalil Ghorbal, Martin Otter, and Marc Pouzet.

Structural Analysis of Multi-Mode DAE Systems.

In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Pittsburgh, USA, April 18-20 2017. ACM.

Gérard Berry.

The constructive semantics of pure esterel, draft, version 3.

Draft book. Available at:
`http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf`, 2002.

Timothy Bourke and Marc Pouzet.

Zélus, a Synchronous Language with ODEs.

In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.

P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice.

Lustre: a declarative language for programming synchronous systems.

In *14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet.

A Conservative Extension of Synchronous Data-flow with State Machines.

In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

References II

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.

Scade 6: A Formal Language for Embedded Critical Software Development.
In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

Pascal Cuoq and Marc Pouzet.

Modular Causality in a Synchronous Stream Language.
In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

N. Halbwachs, P. Raymond, and C. Ratel.

Generating efficient code from data-flow programs.
In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.

Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.
In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.